Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *http://www.jstatsoft.org/*

Dynamic and Interactive R Graphics for the Web: The gridSVG Package

Paul Murrell
The Unversity of Auckland

Simon Potter
The Unversity of Auckland

Abstract

This article describes the **gridSVG** package, which provides functions to convert **grid**-based R graphics to an SVG format. The package also provides a function to associate hyperlinks with components of a plot, a function to animate components of a plot, a function to associate any SVG attribute with a component of a plot, and a function to add JavaScript code to a plot. The last two of these provides a basis for adding interactivity to the SVG version of the plot. Together these tools provide a way to generate dynamic and interactive R graphics for use in web pages.

Keywords: world-wide web, graphics, R, SVG.

1. Introduction

Interactive and dynamic plots within web pages are becomingly increasingly popular, as part of a general trend towards making data sets more open and accessible on the web, for example, GapMinder (Rosling 2008) and ManyEyes (Viegas, Wattenberg, van Ham, Kriss, and McKeon 2007).

The R language and environment for statistical computing and graphics (R Development Core Team 2011) has many facilities for producing plots, and it can produce graphics formats that are suitable for including in web pages, but the core graphics facilities in R are largely focused on *static* plots.

This article describes an R extension package, **gridSVG**, that is designed to embellish and transform a standard, static R plot and turn it into a dynamic and interactive plot that can be embedded in a web page.

1.1. R graphics

Several R packages are being developed to produce interactive and dynamic plots for inclusion in web pages. For example, the **googleVis** package (Gesmann and de Castillo 2011) provides an interface to the Google Visualization API and the **webvis** package (Conway 2010) produces JavaScript code for the protovis (Bostock and Heer 2009) web visualization library. In both cases, the graphics are produced by an external system rather than R. This abandons the powerful graphical facilities that R possesses.

The **gridSVG** package provides a way to add dynamic and interactive features to plots that are produced by R (specifically, plots that are based on the **grid** graphics package; Murrell 2011).

1.2. Vector graphics

It is straightforward to produce a *static* R graphic that can be included in a web page. For example, a plot that is saved as a PNG file can be included via an HTML `<img>` element. However, raster image formats such as PNG are only designed for a specific size and do not scale well.

A better solution is possible with recent web browsers because they now have native support for the SVG format (Eisenberg 2002). This allows *vector* graphics to be used for a web page, which will produce better results because the plot should look good at any scale. The SVG file can be included in a web page via an HTML `<object>` element.

Producing static SVG output from R is a little more complicated than producing PNG output because it depends on the operating system. On Linux and MacOS X, there is an `svg()` function, but on Microsoft Windows the **Cairo** package (Urbanek and Horner 2011) must be installed to provide the `CairoSVG()` function.

Some R extension packages can be used to add some interactivity to an R plot in a PNG format. These make use of the fundamental HTML image map technology whereby regions of a raster image can be identified and have interactive behaviour associated with them. The **iWebPlots** (Chatzimichali and Bessant 2011) package provides functions for producing image maps with tooltips for some standard plot types and the **imagemap** package (Rowlingson 2008) provides similar functionality.

The **gridSVG** package targets the SVG format instead, to allow dynamic and interactive features to be added to R plots in a *vector* format.

1.3. Smooth animation

The **animation** package (Xie 2011) can be used to animate standard R graphics. This works by producing multiple static images, in a raster format, and then either stitching the images together in a movie format, such as MPEG4, or playing the images rapidly one after the other. A movie file can be embedded in a web page using an `<object>` element and the **animation** package provides a convenience function to generate a web page with JavaScript controls for playing a sequence of images.

One limitation of this approach is that the result is a raster image, with the disadvantages mentioned above. It is also not possible to interact with this sort of dynamic plot. Furthermore, the animation is described in a "brute force" method, with every frame having to be

drawn separately.

By targeting the SVG format, the **gridSVG** package allows both dynamic and interactive features at once, in a vector graphics format, and it allows dynamic features to be described more elegantly. For example, in order to make a single point move within a plot, it is only necessary to describe the changing location of that point, and in simple cases only the start and end locations are required.

1.4. Direct interaction

A number of R packages provide GUI toolkits, such as **tcltk** (R Development Core Team 2011), **RGtk2** (Lawrence and Temple Lang 2010), and **gWidgets** (Verzani 2011). With regard to graphics, these provide a way to allow user interaction with a plot via menus, dialogs, buttons, and sliders.

However, these sorts of interfaces do not allow direct interaction with the components of a plot, such as tooltips on data points and drill-down by clicking on a region of a plot.

The **gridSVG** package is designed to allow interaction with individual components of an R plot.

1.5. Extensibility

The **RSVGTipsDevice** package (Plate 2011) provides an R graphics device that saves R plots in an SVG format and allows tooltips and hyperlinks to be associated with different components of the plot. The main limitation of this package is that it is hard-wired for these two sorts of interaction (tooltips and hyperlinks). The package is not designed to be extended by the user for other sorts of interaction.

The **SVGAnnotation** package (Temple Lang 2011) provides a much wider range of facilities for adding dynamic and interactive features to an R plot (in an SVG format). It is possible to add tooltips, animate points, and even link plots (so that clicking on a point in one plot highlights the corresponding point in a separate plot). This package will be discussed in more detail in a later section, once we have established the structure of the **gridSVG** package. For now, it will just be stated that although the **SVGAnnotation** package provides some very powerful functions for adding interaction to some plots, it is not straightforward for a non-expert to extend the package to plots that are not currently supported.

One aim of the **gridSVG** package is to provide general tools that can be easily adapted by non-experts to add interactivity to any plot or indeed any graphic in general that has been produced by R's **grid** graphics system.

In summary, the **gridSVG** package is aimed at producing dynamic and interactive R graphics, in a vector format, for inclusion in web pages, using a transparent mechanism that can be easily adapted and extended by the non-expert user.

2. The gridSVG package

This section gives a brief introduction to the main features of the **gridSVG** package.

The main function in the **gridSVG** package is `gridToSVG()`, which copies the current **grid** scene to an SVG document. An example of a simple **grid** scene, consisting of two rectangles,

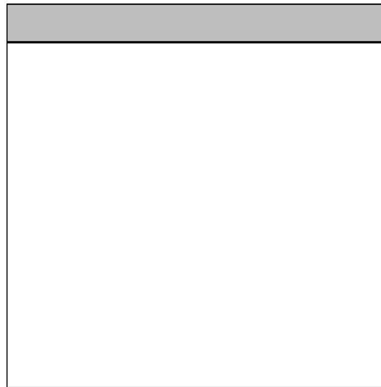

Figure 1: A simple **grid** scene consisting of two rectangles, one above the other.

one above the other, is shown in Figure 1. The code to produce this scene is shown below.

```
R> library("grid")
```

```
R> topvp <- viewport(y=1, just="top", name="topvp",
R+                   height=unit(1, "lines"))
R> botvp <- viewport(y=0, just="bottom", name="botvp",
R+                   height=unit(1, "npc") - unit(1, "lines"))
R> grid.rect(gp=gpar(fill="grey"), vp=topvp, name="toprect")
R> grid.rect(vp=botvp, name="botrect")
```

The following call to `gridToSVG()` creates an SVG document from this scene. The resulting document is called `"gridscene.svg"` and Figure 2 shows this document as it appears in a web browser.

```
R> library("gridSVG")
```

```
R> gridToSVG("gridscene.svg")
```

The current **grid** scene could be produced from raw **grid** calls, as above, or it could be produced from calls to functions from packages that are built on top of **grid**, such as **lattice**, **ggplot2**, **vcd**, and **partykit**. Sections 2.3 and 4 provide some more complex examples.

The result of calling `gridToSVG()` is just a static SVG version of the static **grid** scene, but the **gridSVG** package provides other functions to add dynamic and interactive features.

2.1. Animation

The `grid.animate()` function allows the components of a **grid** scene to be animated. For example, the following code animates the two rectangles in the simple **grid** scene above so that they both shrink to a width of 1 inch over a period of 3 seconds (see Figure 3). The widths begin with a value of 1npc and end with a value of 1in.

Figure 2: The simple **grid** scene from Figure 1 in an SVG format, viewed in Firefox.

```
R> widthValues <- unit(c(1, 1), c("npc", "in"))
R> widthValues

[1] 1npc 1in
```

The component of the **grid** scene that is to be animated is specified by name. For example, the first call to `grid.animate()` below modifies the rectangle called `"toprect"`.

```
R> grid.animate("toprect", width=widthValues, duration=3)
R> grid.animate("botrect", width=widthValues, duration=3)

R> gridToSVG("gridanim.svg")
```

2.2. Interactivity

The `grid.hyperlink()` function associates a hyperlink with a component of a **grid** scene. For example, the following code adds some text to the top rectangle in the simple **grid** scene so that when the text is clicked the web browser will navigate to the R home page (see Figure 4). Again, the component that is to be turned into a hyperlink is specified by name; in this case, it is the text component called `"hypertext"`.

```
R> grid.text("take me there", vp=topvp, name="hypertext")
R> grid.hyperlink("hypertext", "http://www.r-project.org")

R> gridToSVG("gridhyper.svg")
```

The `grid.garnish()` function allows general SVG attributes to be associated with a component of a **grid** scene. For example, the following code adds an event handler to the bottom rectangle in the simple **grid** scene above so that, when the mouse is clicked over the bottom

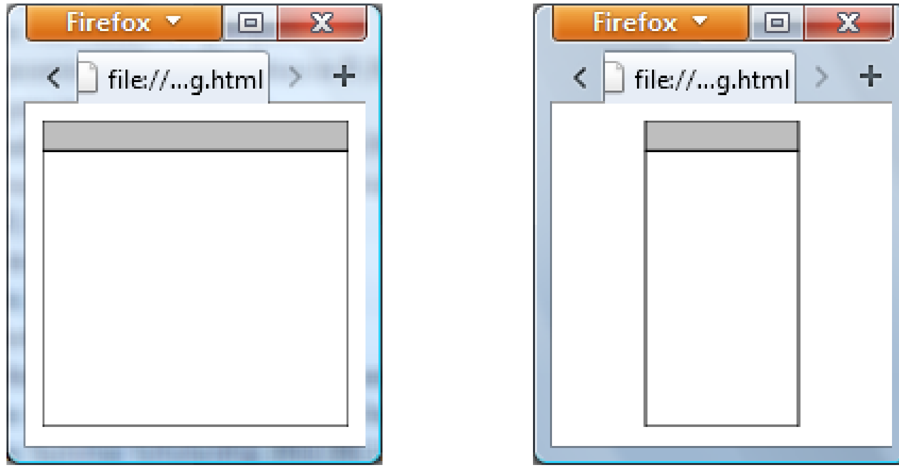

Figure 3: A simple animated **grid** scene. The scene starts off as shown on the left (the same as Figure 2), but then the rectangles shrink over three seconds until they end up only 1 inch wide, as shown on the right.

Figure 4: A simple **grid** scene containing hyperlinked text. If the text is clicked on with the mouse, the web browser will navigate to the R home page.

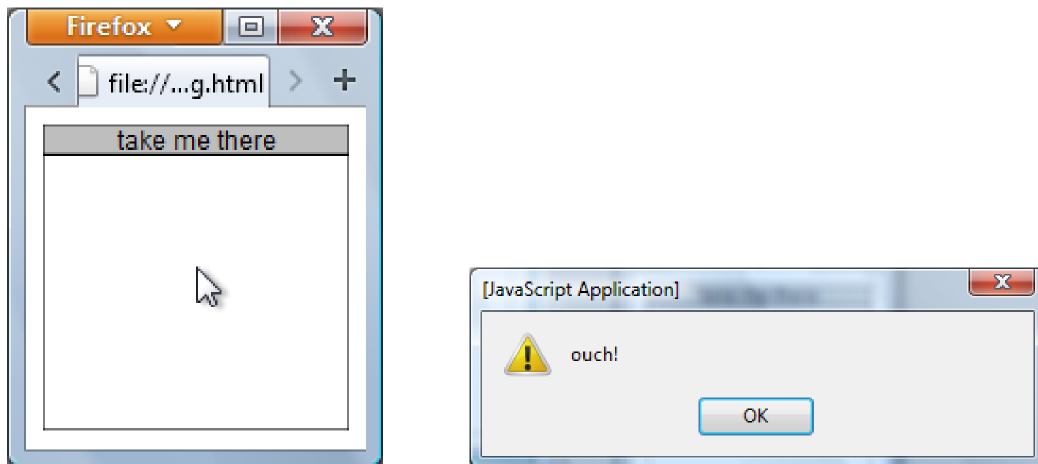

Figure 5: A simple interactive **grid** scene. When the mouse is clicked over the bottom rectangle, an alert dialog pops up.

rectangle, an alert dialog pops up (see Figure 5). The `onmousedown` attribute specifies what action to take when the mouse is clicked and the `pointer-events` attribute makes sure that the rectangle notices mouse clicks within its interior (not just on its drawn border).

```
R> grid.garnish("botrect",
R+              onmousedown="alert('ouch!')",
R+              "pointer-events"="all")

R> gridToSVG("gridmouse.svg")
```

More complex interaction requires adding JavaScript (Flanagan 2006) to the SVG output. This is achieved via the `grid.script()` function, which can be given either JavaScript code as a character vector or the name of a file that contains JavaScript code. For example, the following code adds an event handler to the top rectangle in the simple **grid** scene so that, when the mouse is clicked within the top rectangle, the interior of the rectangle is filled black (see Figure 6). The design of the JavaScript code will be described in Section 3 and Section 4 contains some more sophisticated examples of interaction.

```
R> grid.garnish("toprect",
R+              onmousedown="allblack()",
R+              "pointer-events"="all")
R> grid.script("
R+   allblack = function() {
R+     rect = document.getElementById('toprect.1');
R+     rect.setAttribute('style', 'fill:black');
R+   }")

R> gridToSVG("gridscript.svg")
```

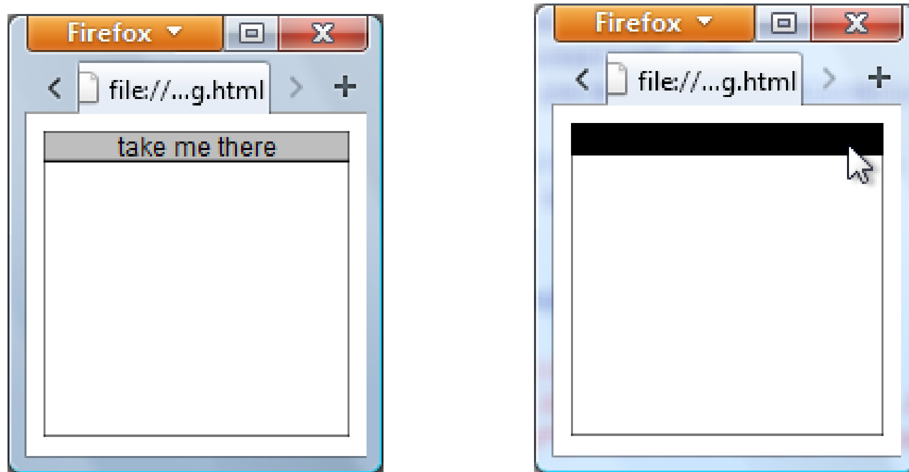

Figure 6: A slightly more complex interactive **grid** scene. When the mouse is clicked over the top rectangle, the top rectangle is filled with black.

2.3. Graphics based on grid

The examples so far have used direct calls to **grid** functions to draw a scene, but the techniques described will also work with plots drawn by packages that are built on **grid**, such as **lattice** (Sarkar 2008), **ggplot2** (Wickham 2009), **vcd** (Meyer, Zeileis, and Hornik 2010), and **partykit** (Hothorn and Zeileis 2011).

For example, the following code generates a multipanel conditioning plot using the **lattice** package.

```
R> library("lattice")
```

```
R> xyplot(Sepal.Length ~ Sepal.Width | Species, iris)
```

The next piece of code uses the **grid.hyperlink()** function to add hyperlinks to the strip labels in that plot.

```
R> grid.hyperlink("plot_01.textr.strip.1.1",
R+                "http://en.wikipedia.org/wiki/Iris_flower_data_set")
R> grid.hyperlink("plot_01.textr.strip.1.2",
R+                "http://en.wikipedia.org/wiki/Iris_virginica")
R> grid.hyperlink("plot_01.textr.strip.2.1",
R+                "http://en.wikipedia.org/wiki/Iris_versicolor")
```

The plot can now be converted to SVG using `gridToSVG()` so that clicking on the strip labels in a web browser will navigate to the relevant pages of Wikipedia (see Figure 7).

```
R> gridToSVG("latticehyper.svg")
```

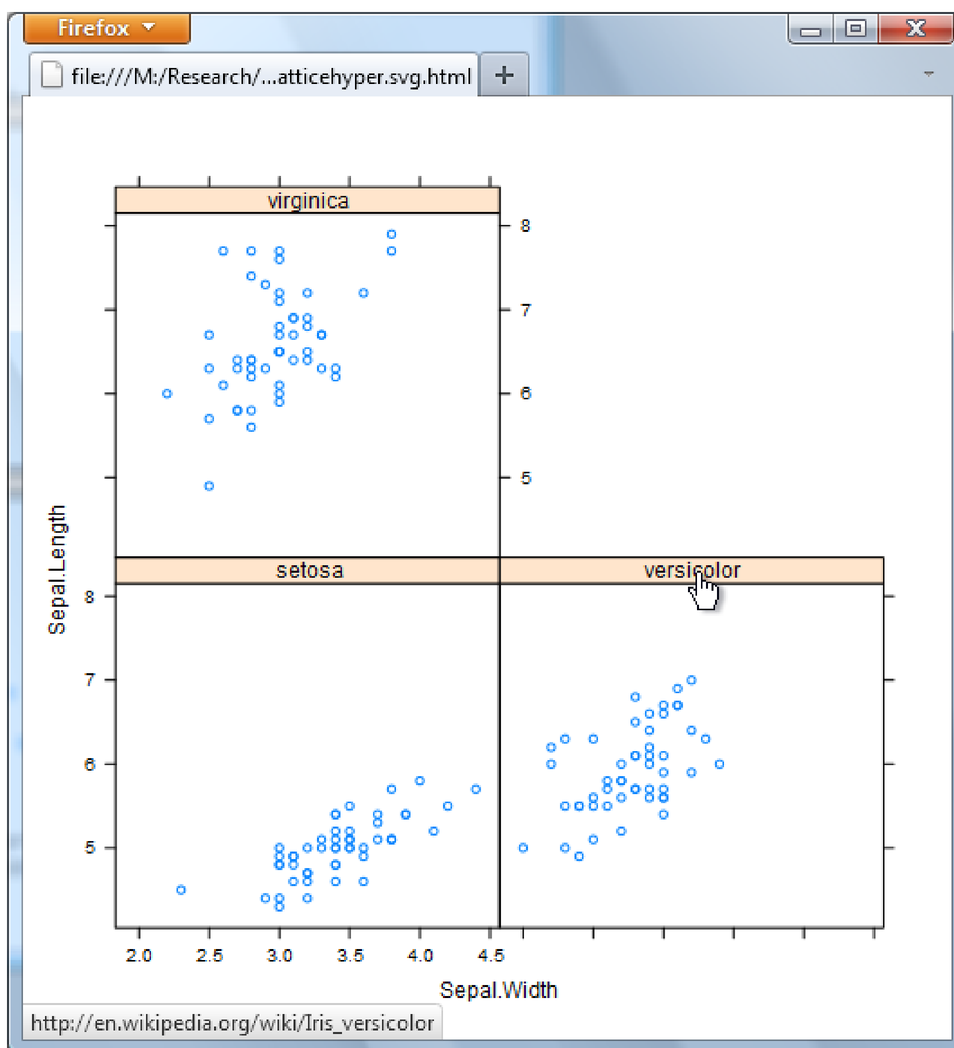

Figure 7: A **lattice** plot with hyperlinks on the strip labels.

3. The design of gridSVG

This section describes some of the details about how the **gridSVG** package works. It is necessary to understand some of these details in order to be able to produce more complex dynamic and interactive plots with **gridSVG**.

3.1. The components of a grid scene

The **gridSVG** package works by looking at what has been drawn on the current page and converting that drawing to SVG. The **gridSVG** package works off the **grid** display list, which is a record of all **grid** output on the current page. This display list consists of **grid** viewports and grobs (graphical objects).

The `grid.ls()` function can be used to list the grobs and viewports in the current scene and the following code demonstrates this for the simple scene from Figure 1. There are two `rect` grobs, called `toprect` and `botrect`, and each rectangle is drawn in a its own viewport (`topvp` and `botvp`, respectively).

```
R> grid.ls(viewports=TRUE, fullNames=TRUE, print=grobPathListing)
```

```
viewport[ROOT]::viewport[topvp] | rect[toprect]
viewport[ROOT]::viewport[botvp] | rect[botrect]
```

3.2. Converting grobs to SVG

The main function in the **gridSVG** package is called `gridToSVG()`. This function takes each grob in the current scene and converts it to one or more SVG elements.

```
R> gridToSVG("gridscene.svg")
```

In the simple **grid** scene shown in Figure 1, each `rect` grob is converted to an SVG `<rect>` element, as shown below.

```
<rect id="toprect.1" x="0" y="129.6" width="144" height="14.4"/>

<rect id="botrect.1" x="0" y="0" width="144" height="129.6"/>
```

Most grobs have an obvious correspondence to an `SVG` element like this (see Table 1), but there are some exceptions. For example, the many different sorts of **grid** line grobs all get converted to an SVG `<polyline>` element. This includes open `xspline` grobs (the smooth curve is stored as a series of short straight lines).

Another exception is the `text` grob, which converts to a series of nested SVG elements. This is necessary in order to get the size and orientation of the text correct. For example, the following **grid** code produces a text grob.

```
R> grid.text("take me there", name="sampleText")
```

Table 1: The SVG elements that basic **grid** grobs are converted into.

grid grob	SVG element
rect	<rect>
circle	<circle>
lines	<polyline>
polyline	<polyline>
segments	<polyline>
xspline	<polyline> or <path>
polygon	<polygon>
path	<path>
raster	<image>
text	<g><g><text><tspan>

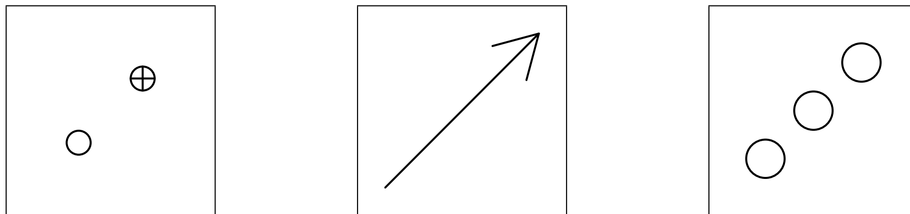

Figure 8: Three simple grid scenes: on the left, two data symbols (one a simple circle and the other a combination of a circle and two lines); in the middle, a line segment with an arrow head on the end; on the right, three circles which are drawn from a single grob.

The `gridToSVG()` function converts this text to SVG elements shown below.

```
<g id="sampleText.1" stroke-width=".1" transform="translate(72, 72) ">
  <g transform="scale(1, -1)">
    <text x="0" y="0" text-anchor="middle">
      <tspan dy="4.31" x="0">take me there</tspan>
    </text>
  </g>
</g>
```

A `points` grob is also handled differently because different data symbols consist of different shapes, so the SVG element that is produced will depend on the choice of data symbol. Some data symbols also consist of more than one shape, for example, a circle with a plus inside it. In those cases, the `points` grob is converted to a `<g>` element with several other elements inside it. For example, the following **grid** code draws two data symbols, one a simple circle and one a combination of a circle and a plus (a vertical line and a horizontal line; see Figure 8).

```
R> pushViewport(viewport())
R> grid.points(1:2/3, 1:2/3, pch=c(1, 10), name="symbols")
```

The `gridToSVG()` function converts these data symbols to the SVG elements shown below.

```
<circle id="symbols.1" cx="48" cy="48" r="4.31"/>

<g id="symbols.2">
  <polyline id="symbols.2.1" points="91.69,96 100.31,96"/>
  <polyline id="symbols.2.2" points="96,91.69 96,100.31"/>
  <circle id="symbols.2.3" cx="96" cy="96" r="4.31"/>
</g>
```

All line grobs in **grid** can have arrow heads at either end. These are implemented in SVG using `<marker>` elements, which are then referred to using `marker-start` and `marker-end` attributes in the `<polyline>` element. The following **grid** code draws a line segment with an arrow at the end (see Figure 8).

```
R> grid.segments(0, 0, 1, 1, arrow=arrow(), name="lineWithArrow")
```

This `segments` grob, with its arrow, is converted to the following SVG code.

```
<defs>
  <marker id="lineWithArrow.1.markerEnd" refX="18" refY="10.39"
          overflow="visible" markerUnits="userSpaceOnUse" markerWidth="18"
          markerHeight="20.78" orient="auto">
    <path d="M 0 0 L 18 10.39 L 0 20.78"/>
  </marker>
</defs>

<polyline id="lineWithArrow.1" points="0,0 144,144"
          marker-end="url(#lineWithArrow.1.markerEnd)"/>
```

3.3. Converting grobs that draw multiple shapes

A single **grid** grob may produce several distinct shapes. For example, the following code creates a single `circle` grob, but three distinct circles are drawn (see Figure 8).

```
R> grid.circle(x=1:3/4, y=1:3/4, r=0.1, name="circles")
```

When this grob is converted to SVG, three distinct `<circle>` elements are created. Because it may be useful to treat the three elements as a coherent group, the conversion nests the three `<circle>` elements within a `<g>` element. The resulting SVG code for this example is shown below.

```
<g id="circles">
  <circle id="circles.1" cx="36" cy="36" r="14.4"/>
  <circle id="circles.2" cx="72" cy="72" r="14.4"/>
  <circle id="circles.3" cx="108" cy="108" r="14.4"/>
</g>
```

For consistency in the SVG code, this `<g>` element is added whether a grob produces multiple shapes or not. For example, the `toprect` grob from Figure 1, which produces only a single `<rect>` element, is actually recorded in SVG as shown below.

```
<g id="toprect">
  <rect id="toprect.1" x="0" y="129.6" width="144" height="14.4"/>
</g>
```

3.4. Converting grob names

Each **grid** grob has a name and this name gets recorded as the `id` attribute of the corresponding SVG element. This conversion is slightly complicated by the fact that a single grob always generates multiple SVG elements (as described in the previous section), so some "mangling" of the original grob name is necessary.

The general rule is that the parent `<g>` element has an `id` attribute that is identical to the original grob name. The children of that `<g>` element have `id` attributes based on the original grob name, but with a numeric suffix appended. The sample SVG code in the previous section demonstrates this idea.

Some cases are more complicated, for example, determining the `id` attributes of `<marker>` elements when converting lines with arrow heads (see Section 3.2), but in all cases the `id` value is based upon the original grob name, with a sensible extension added.

3.5. Converting graphical parameters

The detailed appearance of **grid** grobs—things like colours, line widths, and fonts—is controlled by a set of graphical parameters, such as `col` and `fill` (border and fill colours), `lwd` (line width), and `fontface` and `fontfamily`.

When a **grid** grob is converted to an SVG element, these graphical parameter settings are converted to *presentation attributes* in the SVG element.

There are always default graphical parameter settings in place so that a grob will only specify explicit settings where necessary. This is reflected in SVG, with presentation attributes only recorded when they are explicitly set. A top level `<g>` element establishes a full set of defaults.

For example, in the simple **grid** scene from Figure 1, the `toprect` grob explicitly specifies a grey fill via the `gp` argument.

```
R> grid.rect(gp=gpar(fill="grey"), vp=topvp, name="toprect")
```

The top level `<g>` element and the `<rect>` element for the `toprect` grob are shown below. Many default presentation attribute settings are established in the `<g>` element and the grey fill for the `<rect>` element is recorded as `fill="rgb(190,190,190)"`.

```
<g id="gridSVG"
   fill="none"
   stroke="rgb(0,0,0)"
   stroke-dasharray="none" stroke-width="1px"
   font-size="12px"
```

```
   font-family="Helvetica,Arial,FreeSans,Liberation Sans,Nimbus Sans L,sans-serif"
   opacity="1"
   stroke-linecap="round" stroke-linejoin="round" stroke-miterlimit="10"
   stroke-opacity="1" fill-opacity="0">

  <rect id="toprect.1" x="0" y="129.6" width="144" height="14.4"
        fill="rgb(190,190,190)"
        fill-opacity="1"/>

</g>
```

In all other code samples in this article, these presentation attributes have been removed to aid readability of the SVG code.

3.6. Converting viewports

A **grid** scene consists not only of grobs, but also viewports, which provide contexts for drawing the grobs. The conversion of grobs to SVG makes use of the viewports to determine the correct locations and sizes to use within the SVG elements, but the viewports themselves are also recorded in the SVG code as `<g>` elements, with an `id` attribute set from the viewport name.

If a viewport sets the clipping region, a `<clipPath>` element is added to the SVG output and this is referenced in the `<g>` element for the viewport using the `clip-path` attribute.

We now have enough information to understand the complete SVG code that is produced by `gridToSVG()` for the simple **grid** scene in Figure 1. This SVG code is shown in Figure 9.

3.7. Converting gTrees

A gTree is a grob that can have other grobs as its children. For example, the following code creates a gTree that has a `text` grob and a `rect` grob as its children.

```
R> tg <- textGrob("this is a gTree", name="textchild")
R> rg <- rectGrob(width=grobWidth(tg) + unit(2, "mm"),
R+                height=unit(1, "lines"),
R+                gp=gpar(col=NA, fill="grey"), name="rectchild")
R> textWithBG <- gTree(children=gList(rg, tg), name="gtree")
```

When a gTree is drawn, it draws all of its children. For example, the following code calls the above function to create a gTree and then draws it, which has the effect of drawing both the text grob and the rect grob (see Figure 10).

```
R> grid.draw(textWithBG)
```

The output from `grid.ls()` shows the hierarchical nature of the scene just created.

```
R> grid.ls(fullNames=TRUE)
```

```
<?xml version="1.0" encoding="UTF-8"?>
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="144px" height="144px" version="1.0">
  <g transform="translate(0, 144)  scale(1, -1)">
    <g id="gridSVG" font-size="12px">
      <g id="topvp.1">
        <g id="toprect">
          <rect id="toprect.1" x="0" y="129.6" width="144" height="14.4"/>
        </g>
      </g>
      <g id="botvp.1">
        <g id="botrect">
          <rect id="botrect.1" x="0" y="0" width="144" height="129.6"/>
        </g>
      </g>
    </g>
  </g>
</svg>
```

Figure 9: The SVG code that is generated by `gridToSVG()` from the simple **grid** scene in Figure 1. All `style` attributes have been removed to improve readability.

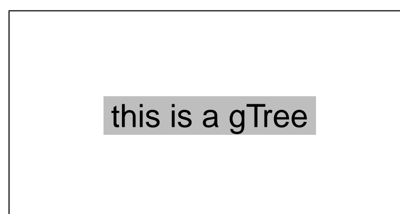

Figure 10: A **grid** scene consisting of a gTree, with a rectangle and a piece of text as its children.

```
rect[toprect]
rect[botrect]
text[sampleText]
segments[lineWithArrow]
gTree[gtree]
  rect[rectchild]
  text[textchild]
```

When a gTree is exported to SVG, the gTree itself creates a `<g>` element, with the child grobs exported as SVG elements within that. The SVG code from the gTree created above is shown below.

```
<g id="gtree">
  <g id="rectchild">
    <rect id="rectchild.1" x="212.21" y="244.8" width="79.59" height="14.4"/>
  </g>
  <g id="textchild">
    <g id="textchild.1" stroke-width=".1" transform="translate(252, 252) ">
      <g transform="scale(1, -1)">
        <text x="0" y="0" text-anchor="middle">
          <tspan dy="4.31" x="0">this is a gTree</tspan>
        </text>
      </g>
    </g>
  </g>
</g>
```

3.8. Fonts

The **gridSVG** package does not embed fonts in the SVG document that it produces. It only records the name of the font that should be used to render text. Because there is no guarantee that a web browser (or the operating system of its host) has access to a particular font, **gridSVG** also records additional font names so that if the first font cannot be found, there are alternatives that can be used.

The collection of font names is called a *font stack* and the **gridSVG** package maintains three font stacks: one for serif fonts, one for sans-serif fonts, and one for monospace fonts. In each case, the final font name in each stack is a generic name, like `"monospace"` so that the web browser should always be able to find something to use.

When text is exported to SVG, the font stack that is recorded is the one that contains the font family used for that text. For example, the following code draws text with a Helvetica font.

```
R> grid.text("font test", gp=gpar(fontfamily="Helvetica"),
R+           name="font")
```

The Helvetica font is part of the sans-serif font stack, so the SVG code records the sans-serif font stack, as shown below (see the `font-family` attribute within the `<text>` element).

```
<g id="font">
  <g id="font.1" stroke-width=".1" transform="translate(252,252) ">
    <g transform="scale(1,-1)">
      <text x="0" y="0" text-anchor="middle"
            font-family="Helvetica,Arial,FreeSans,Liberation Sans,Nimbus Sans L,sans-serif"
            fill="rgb(0,0,0)" fill-opacity="1">
        <tspan dy="4.31" x="0">font test</tspan>
      </text>
    </g>
  </g>
</g>
```

The default font stacks only contain some common font names (on the major operating systems). However, the `setSVGFonts()` function is provided to allow further fonts to be added to the font stacks. The `getSVGFonts()` function can be used to obtain the current font stacks. For example, the following code adds the Inconsolata font to the monospace font stack.

```
R> fontstacks <- getSVGFonts()
R> fontstacks$mono <- c("Inconsolata", fontstacks$mono)
R> setSVGFonts(fontstacks)
```

The new monospace font stack is shown below.

```
R> getSVGFonts()$mono

[1] "Inconsolata"   "Courier"       "Courier New"   "Nimbus Mono L"
[5] "monospace"
```

Again, this does not guarantee that the Inconsolata font will be used to display the text in a browser; that will only happen if the browser has access to the Inconsolata font.

This concludes the discussion of how standard **grid** objects are converted to SVG. The following sections describe how **gridSVG** adds dynamic and interactive features to a **grid** scene and exports those features to SVG code.

3.9. Hyperlinks

The `grid.hyperlink()` function allows hyperlinks to be associated with a component of a **grid** scene. This function works by modifying the appropriate grob to add information about the hyperlink. The modified grob is also given the class `linked.grob` so that it can be handled differently when it is converted to SVG. The conversion of a `linked.grob` to SVG creates an `<a>` (anchor) element around the normal SVG elements that would otherwise be generated.

For example, the following code draws a piece of text and associates a hyperlink with the text so that when the text is clicked with the mouse in a web browser, the browser will navigate to the R home page.

```
R> grid.text("take me there", name="hypertext")
R> grid.hyperlink("hypertext", href="http://www.r-project.org")
```

The `text` grob is now a `linked.grob`.

```
R> grid.get("hypertext")
```

```
linked.grob[hypertext]
```

The `gridToSVG()` function converts this `linked.grob` to a set of `<tspan>`, `<text>`, and `<g>` elements as normal, *plus* an `<a>` element around the outside, as shown below.

```
<a xlink:href="http://www.r-project.org">
  <g id="hypertext">
    <g id="hypertext.1" stroke-width=".1" transform="translate(252, 252) ">
      <g transform="scale(1, -1)">
        <text x="0" y="0" text-anchor="middle">
          <tspan dy="4.31" x="0">take me there</tspan>
        </text>
      </g>
    </g>
  </g>
</a>
```

Placing the SVG anchor around the outside of the normal SVG output from a grob means that we should be able to associate an anchor with any component of a **grid** scene.

3.10. Animation

The **gridSVG** package provides the `grid.animate()` function for animating a **grid** grob. This function works by modifying the appropriate grob to add attributes that contain information about the animation. The modified grob is also given the class `animated.grob` so that it can be handled differently when it is converted to SVG. The conversion of an `animated.grob` to SVG creates an `<animate>` element in addition to the normal SVG elements that would otherwise be generated.

For example, the following code draws a circle on the left of the screen and then animates it so that it moves across to the right of the screen.

```
R> grid.circle(x=.2, r=.1,
R+             gp=gpar(fill="black"), name="oneCircle")
R> grid.animate("oneCircle", x=c(.2, .8))
```

The `circle` grob is now an `animated.grob`.

```
R> grid.get("oneCircle")
```

```
animated.grob[oneCircle]
```

The `gridToSVG()` function converts this grob to a `<circle>` element as normal, *plus* an `<animate>` element, as shown below.

```
<animate xlink:href="#oneCircle.1" attributeName="cx"
         begin="0s" calcMode="linear" dur="1s"
         values="100.8;403.2" repeatCount="1" fill="freeze"/>

<g id="oneCircle">
  <circle id="oneCircle.1" cx="100.8" cy="252" r="50.4"/>
</g>
```

A more complex situation arises when a grob generates more than one shape, and thereby generates more than one SVG element. In this case, animation values can be given for each individual shape by specifying a matrix of animation values. For example, the following code draws two circles, one at the bottom-left of the screen and one at the top-right, then animates the circles so that the bottom-left circle moves to the bottom-right of the screen and the top-right circle moves to the top-left of the screen.

```
R> grid.circle(x=c(.2, .8), y=c(.2, .8),
R+             r=.1, gp=gpar(fill="black"), name="twoCircles")
```

The first column of the matrix is used as animation values for the first circle and the second column is used for the second circle. The first circle will transition from an x-value of `.2` to an x-value of `.8`, while the second circle transitions from `.8` to `.2`.

```
R> animValues <- cbind(c(.2, .8), c(.8, .2))
R> animValues

     [,1] [,2]
[1,]  0.2  0.8
[2,]  0.8  0.2

R> grid.animate("twoCircles", x=animValues)
```

The `gridToSVG()` function converts this grob to two `<circle>` elements plus two `<animate>` elements, as shown below.

```
<animate xlink:href="#twoCircles.1" attributeName="cx"
         begin="0s" calcMode="linear" dur="1s"
         values="100.8;403.2" repeatCount="1" fill="freeze"/>
<animate xlink:href="#twoCircles.2" attributeName="cx"
         begin="0s" calcMode="linear" dur="1s"
         values="403.2;100.8" repeatCount="1" fill="freeze"/>

<g id="twoCircles">
  <circle id="twoCircles.1" cx="100.8" cy="100.8" r="50.4"/>
  <circle id="twoCircles.2" cx="403.2" cy="403.2" r="50.4"/>
</g>
```

It is also possible to specify an animation for an entire set of shapes at once, by specifying TRUE for the group argument. For example, the following code draws two circles which *both* disappear after 1 second.

```
R> grid.circle(x=c(.2, .8), y=c(.2, .8),
R+             r=.1, gp=gpar(fill="black"), name="groupCircles")
R> grid.animate("groupCircles",
R+              visibility=c("visible", "hidden"),
R+              begin=1, duration=0.01,
R+              group=TRUE)
```

The SVG code produced for this example has an <animate> element that refers to the parent <g> element rather than the individual <circle> elements, as shown below.

```
<animate xlink:href="#groupCircles" attributeName="visibility"
         begin="1s" calcMode="linear" dur="0.01s"
         values="visible;hidden" repeatCount="1" fill="freeze"/>

<g id="groupCircles">
  <circle id="groupCircles.1" cx="100.8" cy="100.8" r="50.4"/>
  <circle id="groupCircles.2" cx="403.2" cy="403.2" r="50.4"/>
</g>
```

For some shapes, for example polygons, a single shape requires multiple x-values and multiple y-values. This means that, even when animating a single shape, a vector of values is required at each time point. For the simple case of a single shape of this sort, using a matrix for the animation values will still work because grid.animate() will treat each column of the matrix as a different time point (rather than each column being a different shape).

However, for the case of animating a grob that produces multiple shapes, where each shape requires multiple values at each time point, specifying the animation values via vectors or matrices becomes limiting. Furthermore, the animation examples shown so far have actually only involved simple numeric vectors, but features like the x-location of a grob need to be specified as unit objects, so grid.animate() has been doing some work behind the scenes to extract an appropriate coordinate system for the animation.

A more explicit way to express the animation values is to use the animUnit() function to create an animUnit object. For example, the simple set of animation values for a single circle could be generated with the following code. This specifies a single value for each of two time points.

```
R> animUnit(unit(c(.2, .8), "npc"))

$t1
[1] 0.2npc

$t2
[1] 0.8npc
```

The following code generates a single value for each of two time points, for each of two shapes. The `id` argument to the `animUnit()` function is used to associate different animation values with different shapes.

```
R> animUnit(unit(c(.2, .8, .8, .2), "npc"), id=rep(1:2, each=2))

$id1
$id1$t1
[1] 0.2npc

$id1$t2
[1] 0.8npc


$id2
$id2$t1
[1] 0.8npc

$id2$t2
[1] 0.2npc
```

The following code demonstrates how to generate multiple values for each time point, by specifying the `timeid` argument, which associates different animation values with different time points.

```
R> animUnit(unit(c(.2, .8, .8, .2), "npc"), timeid=rep(1:2, each=2))

$t1
[1] 0.2npc 0.8npc

$t2
[1] 0.8npc 0.2npc
```

By specifying *both* `timeid` and `id` arguments, it is possible to specify multiple animation values at each time point for different shapes, as shown below.

```
R> animUnit(unit(c(.2, .8, .8, .2, .1, .9, .9, .1), "npc"),
R+          timeid=rep(rep(1:2, each=2), 2),
R+          id=rep(1:2, each=4))

$id1
$id1$t1
[1] 0.2npc 0.8npc

$id1$t2
[1] 0.8npc 0.2npc
```

```
$id2
$id2$t1
[1] 0.1npc 0.9npc

$id2$t2
[1] 0.9npc 0.1npc
```

Not all features of a grob are locations or dimensions (which require `unit` values), so there is also an `animValue()` function for generating sets of non-`unit` animation values. The vignette `"animation"` in the **gridSVG** package describes these functions in more detail.

3.11. Interactivity

The general mechanism for adding interactivity to a **grid** scene involves associating JavaScript code (Flanagan 2006) with a component of the **grid** scene. This usually requires two steps: adding JavaScript code to the scene and specifying which component of the scene will call that JavaScript code (and when).

The first step is straightforward. The `grid.script()` function adds a `script.grob` object to the current **grid** scene (nothing is drawn on screen, but the grob is added to the **grid** display list). The JavaScript code for the `script.grob` object can be specified as a character vector or as the name of a file that contains JavaScript code. For example, the following code adds JavaScript code to a scene from the file called `"script.js"`.

```
R> grid.script(filename="script.js", name="script")
```

The `gridToSVG()` function generates a `<script>` element from the `script.grob` object. By default, if the JavaScript code is from an external file, the `<script>` element just provides a reference to the external file, as shown below.

```
<script type="text/ecmascript" id="script" xlink:href="script.js">
</script>
```

If the JavaScript code is specified as a character vector or, if the code is in an external file, but the `inline` argument is set to `TRUE`, then the JavaScript code is embedded in the SVG document. For example, the following code creates a simple script that will be directly embedded in the SVG output.

```
R> grid.script("document.onload = alert('hi');",
R+             name="embeddedScript")
```

The resulting SVG code is shown below. When the SVG document is loaded by a browser, an alert dialog will immediately pop up.

```
<script type="text/ecmascript" id="embeddedScript"><![CDATA[
document.onload = alert('hi');
  ]]></script>
```

The second step in adding interactivity to a **grid** scene is to specify which component of the scene will run the JavaScript code (and when). This is achieved with the `grid.garnish()` function, which works by modifying a grob to add attributes that specify when the grob will call JavaScript code. The modified grob is also given the class `garnished.grob` so that it can be handled differently when it is converted to SVG. The conversion of a `garnished.grob` to SVG creates the normal SVG elements that would otherwise be generated, but adds extra SVG attributes to the elements.

For example, the following code draws a circle in the middle of the screen and then adds an attribute so that when the circle is clicked with the mouse in a web browser, an alert dialog will pop up.

```
R> grid.circle(r=.1, gp=gpar(fill="black"), name="clickme")
R> grid.garnish("clickme",
R+              onmousedown="alert('ouch!')",
R+              "pointer-events"="all")
```

The `gridToSVG()` function converts the garnished grob to a `<circle>` element within a `<g>` element as normal, but with `onmousedown` and `pointer-events` attributes added to the `<g>` element, as shown below.

```
<g id="clickme" onmousedown="alert('ouch!')" pointer-events="all">
  <circle id="clickme.1" cx="252" cy="252" r="50.4"/>
</g>
```

The following code shows how `grid.script()` and `grid.garnish()` can be combined so that a component of a **grid** scene can be set up to call JavaScript code from a script that has been added to the scene. In this case, a `circle` grob will call the JavaScript function `allwhite()` when the mouse is clicked within the circle and the `allwhite()` function is defined in a `script.grob` object within the scene.

```
R> grid.circle(r=.1, gp=gpar(fill="black"), name="clickme")
R> grid.garnish("clickme", onmousedown="allwhite()")
R> grid.script("
R+   allwhite = function() {
R+     circle = document.getElementById('clickme.1');
R+     circle.setAttribute('style', 'fill:white');
R+   }", name="allwhite")
```

The SVG generated from this scene is shown in Figure 11. An important feature of the JavaScript code is that it refers to the relevant SVG element by name (`"clickme.1"`). We will return to this point in Section 5.

Another important feature of the code in Figure 11 is that the `onmousedown` attribute has been added to the `<g>` element parent, not the `<circle>` element itself. In some circumstances, this is very useful. For example, the following code draws three circles with one **grid** call and then associates an interaction with all three circles at once.

```
<?xml version="1.0" encoding="UTF-8"?>
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="504px" height="504px" version="1.0">
  <g transform="translate(0, 504)  scale(1, -1)">
    <g id="gridSVG" font-size="12px">
      <g id="clickme" onmousedown="allwhite()">
        <circle id="clickme.1" cx="252" cy="252" r="50.4"/>
      </g>
      <script type="text/ecmascript" id="allwhite"><![CDATA[

  allwhite = function() {
    circle = document.getElementById('clickme.1');
    circle.setAttribute('style', 'fill:white');
  }
  ]]></script>
    </g>
  </g>
</svg>
```

Figure 11: The SVG code that is generated by `gridToSVG()` from a **grid** scene consisting of a garnished `circle` grob and a `script.grob`.

```
R> grid.circle(x=1:3/4, r=.1, gp=gpar(fill="black"),
R+             name="threeCircles")
R> grid.garnish("threeCircles",
R+              onmousedown="alert('ouch')")
```

The SVG version of this scene is shown below, with three `<circle>` elements nested within a `<g>` element. Because the `onmousedown` attribute is added to the `<g>` parent of the the `<circle>` elements, an alert dialog will pop up when *any* of the three circles is clicked with the mouse.

```
<g id="threeCircles" onmousedown="alert('ouch')">
  <circle id="threeCircles.1" cx="126" cy="252" r="50.4"/>
  <circle id="threeCircles.2" cx="252" cy="252" r="50.4"/>
  <circle id="threeCircles.3" cx="378" cy="252" r="50.4"/>
</g>
```

However, it is also possible to make `grid.garnish()` add attributes to the child elements rather than to the parent `<g>`. For example, the following code again draws three circles, but then associates a different interaction with each circle, by specifying three values for the `onmousedown` attribute and setting the `group` argument to `FALSE`.

```
R> grid.circle(x=1:3/4, r=.1, gp=gpar(fill="black"),
R+             name="diffCircles")
```

```
R> grid.garnish("diffCircles",
R+              onmousedown=c("alert('click me!')",
R+                            "alert('no, click me!')",
R+                            "alert('no, no, click me!')"),
R+              group=FALSE)
```

The SVG version of this scene is shown below, with the `onmousedown` attributes set on `<circle>` elements rather than on the `<g>` element.

```
<g id="diffCircles">
  <circle id="diffCircles.1" cx="126" cy="252" r="50.4"
          onmousedown="alert('click me!')"/>
  <circle id="diffCircles.2" cx="252" cy="252" r="50.4"
          onmousedown="alert('no, click me!')"/>
  <circle id="diffCircles.3" cx="378" cy="252" r="50.4"
          onmousedown="alert('no, no, click me!')"/>
</g>
```

3.12. Custom grobs

It is possible to create a new class of grob, typically when the grob has to perform calculations every time it is drawn. For example, the following code defines a new `boxedtext` class that consists of nothing but a label.

```
R> bt <- grob(label="this is a label", name="bt",
R+             cl="boxedtext")
```

This grob will create a `text` and `rect` grob every time it is drawn, which is one way to make sure that the `rect` grob is always the right size for the text. This is achieved by writing a `drawDetails()` method for the `boxedtext` class, as shown below.

```
R> btgrob <- function(x) {
R+     tg <- textGrob(x$label, name="text")
R+     rg <- rectGrob(width=grobWidth(tg) + unit(2, "mm"),
R+                    height=unit(1, "lines"),
R+                    gp=gpar(col=NA, fill="grey"), name="box")
R+     gTree(children=gList(rg, tg), name=x$name)
R+ }

R> drawDetails.boxedtext <- function(x, ...) {
R+     grid.draw(btgrob(x))
R+ }
```

Whenever the `boxedtext` grob is drawn, the `drawDetails()` method is called. The following code draws `bt` and then edits it to change the label and, because the `text` and `rect` grobs are recreated each time, the `rect` grob automatically scales to accommodate the text (see Figure 12).

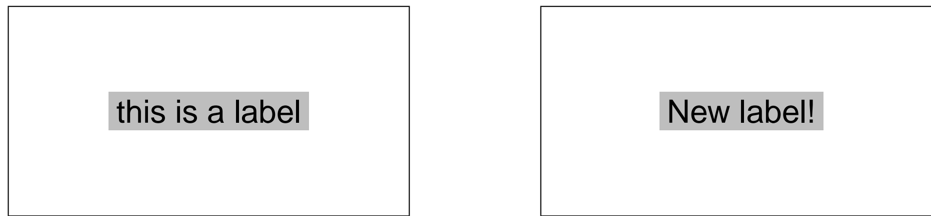

Figure 12: A **grid** scene consisting of a new `boxedtree` grob (on the left). On the right, the `boxedtree` grob has been edited to change the label (and the rectangle has automatically scaled to fit the new label).

```
R> grid.draw(bt)
R> grid.edit("bt", label="New label!")
```

This sort of new grob class presents a problem for **gridSVG** because it does not know how to convert the `boxedtext` grob (which only consists of a label) into SVG code. The solution is to define a `primToDev()` method for the `boxedtext` grobs. This is the generic function that converts grobs into SVG code and a new method can be written quite easily by generating standard **grid** grobs and calling `primToDev` on those grobs (much like a `drawDetails()` method). The following code does this for the `boxedtext` class.

```
R> primToDev.boxedtext <- function(x, ...) {
R+     primToDev(btgrob(x), ...)
R+ }
```

With that method defined, **gridSVG** will export SVG code based on the `text` and `rect` grobs produced by the `btgrob()` function, as shown below.

```
<g id="bt">
  <g id="box">
    <rect id="box.1" x="214.49" y="244.8" width="75.03" height="14.4"/>
  </g>
  <g id="text">
    <g id="text.1" stroke-width=".1" transform="translate(252, 252) ">
      <g transform="scale(1, -1)">
        <text x="0" y="0" text-anchor="middle">
          <tspan dy="4.31" x="0">this is a label</tspan>
        </text>
      </g>
    </g>
  </g>
</g>
```

In addition to the generic function `primToDev()`, there are also generic functions `animate()` and `garnish()` to allow new grob classes to be animated and garnished. The `"extensibility"` vignette in the **gridSVG** package describes these functions in more detail.

4. Examples

This section describes two, more advanced examples of using the **gridSVG** package.

4.1. Animations for teaching

Wild, Pfannkuch, Regan, and Horton (2011) describe a number of animations to be used in developing students' intuition about inference. One example (Figure 10 of Wild *et al.* 2011) shows a barplot of samples from a single categorical variable. The bars transition from one sample to the next and leave a "footprint" consisting of a horizontal line at the top of the bar.

To produce an SVG version of this animation using **gridSVG**, we start with data that consists of a list of counts for each of five categories.

```
R> samples[1:3]

[[1]]

 bike   bus   car other  walk
    8    21    42     3    26

[[2]]

 bike   bus   car other  walk
    5    28    34     2    31

[[3]]

 bike   bus   car other  walk
    6    31    35     6    22
```

The following code generates a **lattice** barplot from the first set of sample data to provide the basis of the animation (see Figure 13).

```
R> boxWidth <- unit(2, "cm")
R> barchart(samples[[1]],
R+          col=rgb(0,0,1,.2),
R+          ylim=c(0, max(sapply(samples, max))),
R+          horizontal=FALSE,
R+          box.width=boxWidth)
```

The next code shows how the heights of the bars can be animated using the **gridSVG** package. This code identifies the grob to animate, which is a `rect` grob that draws the five bars in the plot, and provides a set of animation values for the heights of those bars (a matrix with five columns; each column provides animation values for one bar).

```
R> time <- 10
R> grid.animate("plot_01.barchart.rect.panel.1.1",
```

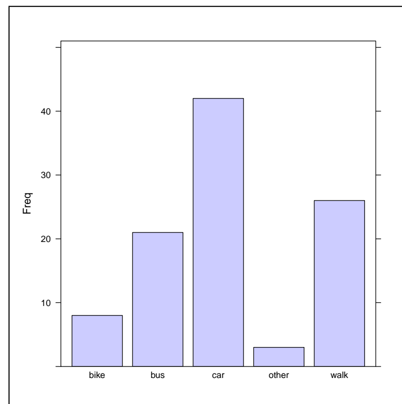

Figure 13: A **lattice** barplot that provides the basis for an animated barplot (see Figure 15).

```
R+              height=do.call("rbind", samples),
R+              duration=time, interpolate="discrete")
```

The "footprints" left behind after each step in the animation are horizontal line segments. The following code adds these segments to the basic plot (see Figure 14).

```
R> trellis.focus("panel", 1, 1)
R> count <- 1
R> samplefun <- function(samp) {
R+     grid.segments(unit(1:5, "native") - 0.5*boxWidth,
R+                   unit(samp, "native"),
R+                   unit(1:5, "native") + 0.5*boxWidth,
R+                   unit(samp, "native"),
R+                   gp=gpar(lwd=2, col=rgb(0,0,1,.2)),
R+                   name=paste("sample.line", count, sep="."))
R+     count <<- count + 1
R+ }
R> lapply(samples, samplefun)
R> trellis.unfocus()
```

The footprints are made to appear at the appropriate times by animating their `visibility` attribute, as shown by the following code.

```
R> for (i in 1:n) {
R+     grid.garnish(paste("sample.line", i, sep="."),
R+                  visibility="hidden")
R+     grid.animate(paste("sample.line", i, sep="."),
R+                  visibility=c("hidden", "visible"),
R+                  group=TRUE,
R+                  begin=i/n*time,
R+                  duration=.01)
R+ }
```

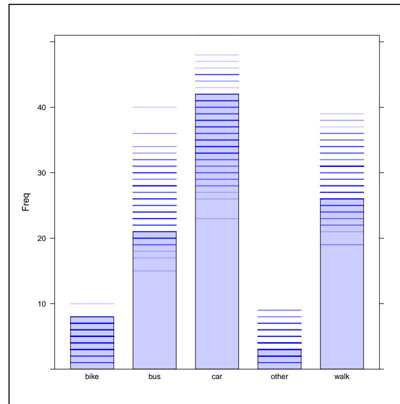

Figure 14: The **lattice** barplot from Figure 13, with horizontal line segments added. These line segments will be animated to leave "footprints" in the animated barplot (see Figure 15).

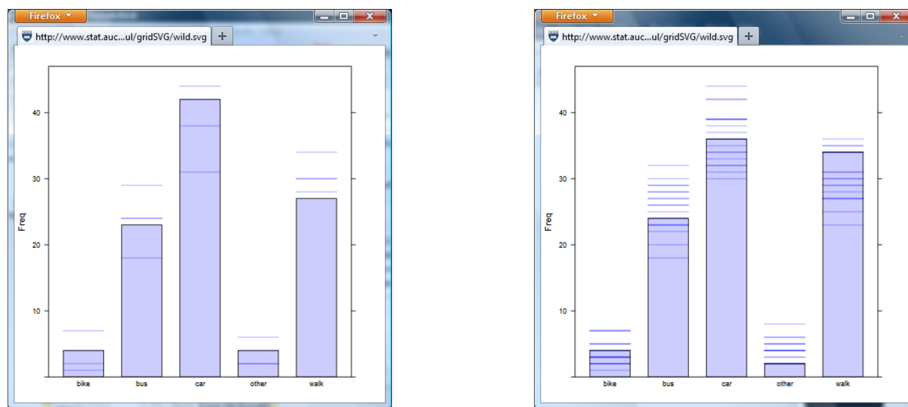

Figure 15: Some snapshots from an animation showing barplots of samples from a single categorical variable.

Figure 15 shows snapshots of the animation and the live SVG file, plus full R code, is available with the online resources for this paper.

4.2. A grid object browser

One of the obstacles to overcome when animating or garnishing a **grid** scene is determining the name of the grob that we wish to animate or garnish. For example, the following code generates a relatively complex plot using **ggplot2** (see Figure 16).

```
R> library("ggplot2")

R> qplot(disp, mpg, data=mtcars) + facet_wrap(~ cyl)
```

This plot has many components (over 100). The first few are shown below (the indenting of some of the lines in that display indicates that the grobs are collected together in a gTree).

```
R> grid.ls()
```

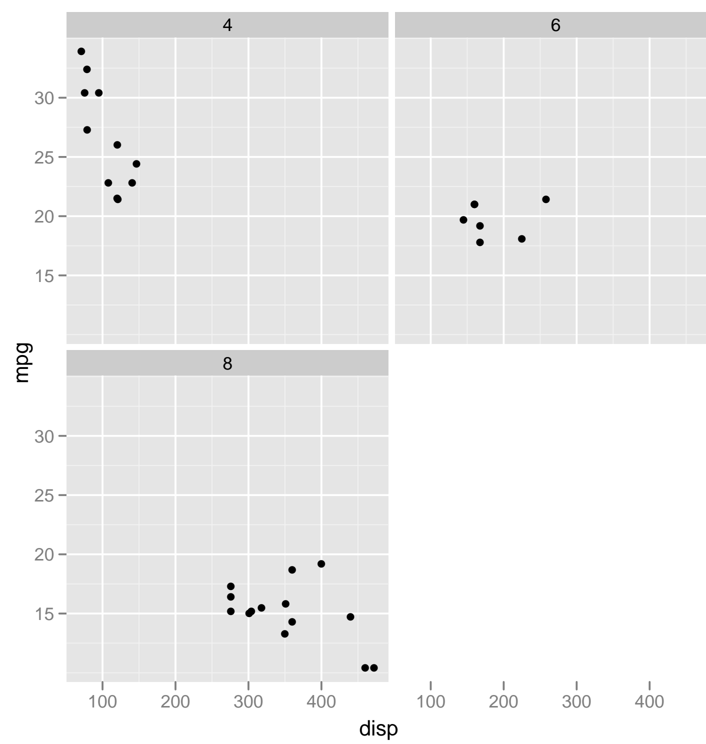

Figure 16: A relatively complex **ggplot2** plot.

```
GRID.gTree.952
  plot.background.rect.951
  plot.title.text.945
  axis.title.x.text.947
  axis.title.y.text.949
```

One way to help explore the many grobs in this plot is to generate an SVG version of the plot with tooltips so that when the mouse hovers over a grob a tooltip is displayed to show the name of the grob. The following code performs this task. First of all, the names of all "leaf" grobs are obtained (all grobs that are not gTrees).

```
R> grobs <- grid.ls()
R> names <- grobs$name[grobs$type == "grobListing"]
```

Now we garnish all of those grobs so that they will respond to the mouse by showing a tooltip.

```
R> for (i in unique(names)) {
R+     grid.garnish(i,
R+                  onmouseover=paste("showTooltip(evt, '", i, "')"),
R+                  onmouseout="hideTooltip()")
R+ }
```

Finally, we add a script file that contains the JavaScript code to draw the tooltips and export the whole scene to SVG. A snapshot of the result, as viewed in a browser, is shown in Figure 17.

```
R> grid.script(filename="tooltip.js")
R> gridToSVG("qplotbrowser.svg")
```

A live version of this plot is available with the online resources for this paper.

5. Discussion

The **gridSVG** package is not the only package that provides tools for generating dynamic and interactive R graphics for inclusion in web pages. This section looks more closely at where the approach taken in **gridSVG** provides advantages over other approaches and where it is weaker than other approaches.

5.1. Avoiding the R graphics engine

An important feature of the **gridSVG** package is that it works directly from the **grid** display list. This is in contrast to packages that provide an R graphics device, like **RSVGTipsDevice**, which work from the graphics calls made by the R graphics engine. Figure 18 shows a diagram of this difference.

The significance of this approach is that **gridSVG** is working with much richer information. The **grid** display list contains a higher-level description of the current scene than the R graphics engine. For example, the following code draws a set of three data symbols (see Figure 19).

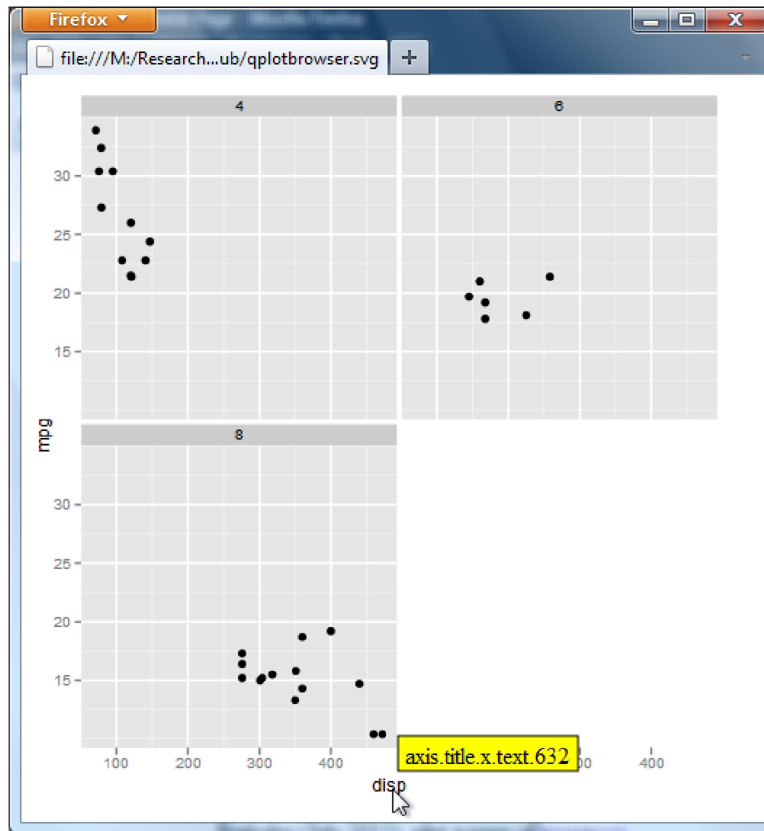

Figure 17: An SVG version of the plot in Figure 16 with tooltips added to show the name of the grob over which the mouse is hovering.

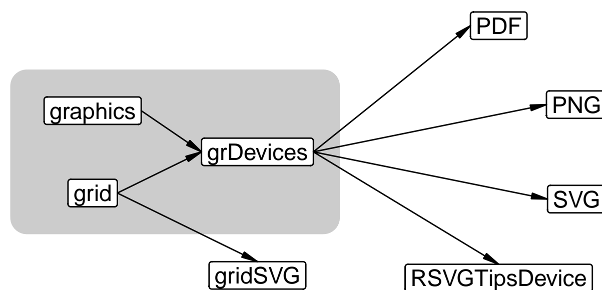

Figure 18: The organisation of graphics packages, which shows that **gridSVG** works directly from the **grid** display list whereas graphics devices work from the R graphics engine (the **grDevices** package).

Figure 19: A set of three data symbols (left) is recorded as a single named grob on the **grid** display list, but gets reduced to 9 separate shapes in the R graphics engine.

```
R> grid.points(1:3/4, 1:3/4, pch=10, name="points")
```

On the **grid** display list, these data symbols are represented as a single grob, called `points`, but the R graphics engine breaks them into 9 separate unnamed shapes (3 circles, 3 vertical lines, and 3 horizontal lines). In the latter case, it is very difficult to identify the 9 shapes as a coherent set, but by working off the **grid** display list, it is straightforward for the **gridSVG** package to identify the data symbols, which makes it possible to animate or garnish the data symbols as a coherent set of shapes.

Because **gridSVG** has richer information to work with, there are more things that it can (easily) do and, at the same time, **gridSVG**'s internal algorithms can be relatively straightforward. That in turn means that the low-level infrastructure of **gridSVG** can be exposed and explained to the user (as shown in Section 3), which means that the user can participate in the development of new uses for **gridSVG**. This is also reflected in the small set of fundamental **gridSVG** functions (`grid.hyperlink()`, `grid.animate()`, `grid.garnish()`, and `gridToSVG()`).

5.2. Generating SVG code directly

Another consequence of working directly off the **grid** display list is that **gridSVG** has complete control over the SVG code that it generates. This is in contrast to the **SVGAnnotation** package, which works off the SVG code that is generated by the R graphics engine. Figure 20 shows a diagram of this difference.

Two main advantages flow from this: the structure of the SVG code that **gridSVG** produces can reflect the structure of the original image; and the SVG elements that **gridSVG** produces have `id` attributes that reflect the names of the grobs in the original scene. Together, these make it much easier to work with the SVG code that is produced by **gridSVG**. Figure 9, which shows the SVG code generated by **gridSVG** for the simple **grid** scene in Figure 1, demonstrates both of these ideas.

By way of comparison, Figure 21 shows the SVG code that the R graphics engine produces from the simple **grid** scene in Figure 1. The **SVGAnnotation** package has to work much harder to identify meaningful components of the original scene from the SVG because the structure of the code does not relate as well to the structure of the original scene and because there are

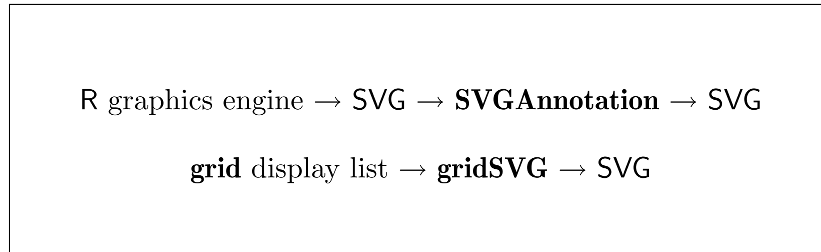

Figure 20: A comparison of **gridSVG**, which works off the **grid** display list and produces SVG directly, and **SVGAnnotation**, which works with the SVG that is produced by the R graphics engine and produces new SVG code from that.

no useful `id` attributes to label components. As a specific example, the two rectangles in the original scene in Figure 1 are represented by four `path` elements in the SVG code in Figure 21.

The predictable structure and useful labelling of the SVG code that is produced by **gridSVG** makes it possible to convey the structure (as shown in Section 3), which makes it possible for non-expert users to work with the SVG code that is produced by **gridSVG**. This is important in understanding the `grid.animate()` and `grid.garnish()` functions *and* in writing JavaScript code to implement interactive features of a scene *and* in working with the SVG code more generally (see Section 5.4).

5.3. Generality

The two previous sections have emphasized that it is not enough to be able to export SVG output from R graphics; a great deal can be gained if that SVG output is also sensibly structured and labelled.

The **gridSVG** approach provides a general solution to exporting SVG output in the sense that it will export the structure and labelling of *any* **grid** scene. As long as the **grid** scene has a clear hierarchical structure (as in **ggplot2** plots) or a clear labelling scheme (as in **lattice** plots), that structure and labelling will be translated to the SVG document that **gridSVG** creates.

This is a significant step forward from packages that work off the R graphics engine because, in that case, almost all structure and labelling from the original scene is lost. The solutions provided by these packages cannot be as general because the package has to be told what functions were used to draw a scene in order to know how to recover structure from the neutered information that is provided by the R graphics engine.

For example, the **SVGAnnotation** package has to be "trained" to handle the output from different sorts of plots. An impressive amount of work has been done to provide useful high-level functions for a range of plots already, but expanding the scope of the package requires extra effort *and* detailed expert knowledge.

The **iWebPlots** package has a similar problem because it has to match the calculations done by the original plot in order to reverse engineer the locations of elements of the plot within a raster image. This means not only training the package for each different sort of plot, but it also limits the components that can be made interactive. For example, it is difficult to recover the positions of axis labels within a raster image.

```
<?xml version="1.0" encoding="UTF-8"?>
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     width="144pt" height="144pt" viewBox="0 0 144 144" version="1.1">
  <defs>
    <clipPath id="clip0">
      <rect width="144" height="144"/>
    </clipPath>
    <clipPath id="clip1">
      <path d="M 0 0 L 145 0 L 145 145 L 0 145 Z M 0 0 "/>
    </clipPath>
    <clipPath id="clip2">
      <path d="M 0 0 L 145 0 L 145 145 L 0 145 Z M 0 0 "/>
    </clipPath>
  </defs>
  <g id="surface0" clip-path="url(#clip0)">
    <rect x="0" y="0" width="144" height="144"/>
    <g clip-path="url(#clip1)" clip-rule="nonzero">
      <path d="M 0 14.399994 L 144 14.399994 L 144 0 L 0 0 Z M 0 14.399994 "/>
      <path d="M 0 14.399994 L 144 14.399994 L 144 0 L 0 0 Z M 0 14.399994 "
            transform="matrix(1,0,0,1,0,0)"/>
    </g>
    <g clip-path="url(#clip2)" clip-rule="nonzero">
      <path d="M 0 144 L 144 144 L 144 14.399994 L 0 14.399994 Z M 0 144 "/>
      <path d="M 0 144 L 144 144 L 144 14.399994 L 0 14.399994 Z M 0 144 "
            transform="matrix(1,0,0,1,0,0)"/>
    </g>
  </g>
</svg>
```

Figure 21: The SVG code that is generated by the Cairo-based `svg()` graphics device from the simple **grid** scene in Figure 1. All `style` attributes have been removed to improve readability.

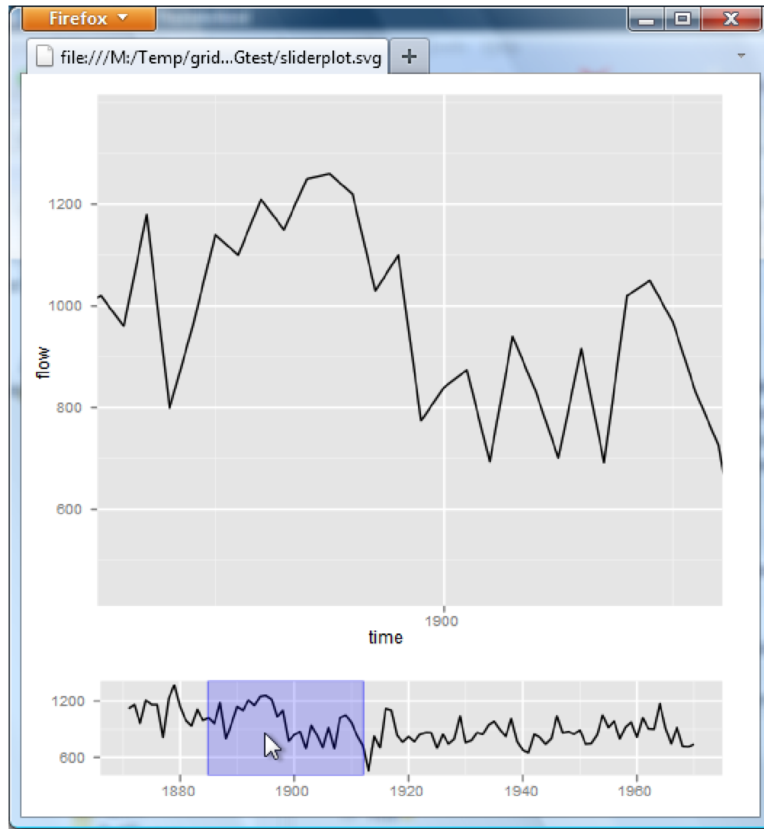

Figure 22: A snapshot of an interactive **ggplot2** scene. The user slides the blue rectangle in the lower plot and the upper plot shows a zoomed view corresponding to the blue rectangle.

5.4. The power of XML

The fact that **gridSVG** produces SVG code, and the fact that the SVG code is predictable, coherently structured, and labelled, means that there are significant opportunities for working with an SVG document after it has been produced by **gridSVG**.

This opportunity arises because SVG is a dialect of XML (Harold and Means 2004) and there are extremely powerful tools for working with XML code, such as XPath and XPointer (Simpson 2002).

As an example, we will look at some of the set up of an interactive **ggplot2** plot. The scene consists of two plots, one above the other, where the upper plot shows a zoomed region of the lower plot. The lower plot has a "thumb" rectangle which the user can slide to determine which region of the lower plot is shown in the upper plot (a snapshot of this interactive scene is shown in Figure 22).

The data used in this scene is a data frame version of the `Nile` time series and the plot is just a simple line plot.

```
R> df <- data.frame(time=as.numeric(time(Nile)),
R+                  flow=as.numeric(Nile))
R> thePlot <- ggplot(df, aes(x=time, y=flow)) +
```

```
R+            geom_line()
R> botPlot <- thePlot
```

The arrangement of the plots is achieved using **grid** viewports and an SVG version is generated with the code below (see top plot in Figure 23).

```
R> doplot <- function() {
R+     pushViewport(viewport(y=1, height=.8, just="top",
R+                           name="topvp"))
R+     print(thePlot, newpage=FALSE)
R+     upViewport()
R+     pushViewport(viewport(y=0, height=.25, just="bottom",
R+                           name="bottomvp"))
R+     print(botPlot, newpage=FALSE)
R+     upViewport()
R+ }
```

```
R> pdf(width=4, height=4)
R> doplot()
R> gridToSVG("normalplot.svg")
R> dev.off()
```

A zoomed version of the scene is generated by drawing the same scene on a much wider device (see the middle plot in Figure 23).

```
R> pdf(width=12, height=4)
R> doplot()
R> gridToSVG("wideplot.svg")
R> dev.off()
```

Now we can begin to take advantage of the fact that the SVG code is XML code. The **XML** package (Temple Lang 2010) provides access to several of the XML processing tools.

```
R> library("XML")
```

The first step is to read the normal scene and the zoomed scene into R.

```
R> normalSVG <- xmlParse("normalplot.svg")
R> wideSVG <- xmlParse("wideplot.svg")
```

Next, we identify the section of SVG code that corresponds to the data region in the top plot of the normal scene. This makes use of XPath to find the `<g>` element with an `id` attribute of `panel-3-3.1` (determined by inspecting the SVG code).

```
R> normalPlotSVG <-
R+     getNodeSet(normalSVG,
R+                "//svg:g[@id='panel-3-3.1']",
R+                c(svg="http://www.w3.org/2000/svg"))[[1]]
```

We can identify the contents of the same region in the zoomed scene with similar code.

```
R> widePlotSVG <-
R+     getNodeSet(wideSVG,
R+                "//svg:g[@id='panel-3-3.1']/svg:g[@id='panel-3-3']",
R+                c(svg="http://www.w3.org/2000/svg"))[[1]]
```

Now all we have to do is remove the original data region and replace it with the zoomed data region. The following code does that and writes out the result to a new SVG document (see the bottom plot in . Figure 23).

```
R> removeChildren(normalPlotSVG, "g")
R> addChildren(normalPlotSVG, widePlotSVG)
R> saveXML(normalSVG, file="customplot.svg")
```

The complete interactive plot, plus R code, is available with the online resources for this paper.

5.5. Drawbacks

The first major limitation of the **gridSVG** package is that it can only export **grid** scenes. It will completely ignore any drawing done by the traditional graphics system (the **graphics** package). This is not a limitation suffered by the packages **SVGAnnotation**, **RSVGTipsDevice**, or **iWebPlots**.

A secondary issue is that the usefulness of the package depends on grobs being sensibly named. This is fortunately not an issue if we call **grid** functions directly to draw a scene (because we can simply supply names ourselves), and the **lattice** package and (to a lesser extent) the **ggplot2** package have named most of the grobs that they create when drawing. However, there are other packages that do not name grobs and that will limit what can be done to animate or interact with scenes drawn by those packages.

Another major weakness of the **gridSVG** approach is that it is focused on producing self-contained SVG documents; the document has to contain *all* of the data and graphics required for any animation or interaction. This has several negative consequences: there is a limit to how much interactivity can be added to a plot, because the document has to contain all possible graphics that might be generated in response to user input, and the documents that are produced tend to be large. The verbosity of the SVG format does not help with the latter problem (though compression to an SVGZ format can be very effective).

It is in theory possible to have R in the background, either on a web server or embedded in the web browser, so that the content of the SVG document can be generated or updated on-the-fly, but the generation of scenes is almost certainly too slow to provide sufficiently responsive interaction. This approach is never going to compete with interactive systems like **iplots** (Urbanek and Wichtrey 2010) or GGobi (Swayne, Cook, Temple Lang, and Buja 2010).

Another issue with the **gridSVG** approach is that, in order to implement any non-trivial interaction, we have to write JavaScript code. It also helps to have an basic understanding of the structure of SVG code. These place an additional burden on the user, although some relief may be provided if a body of generic JavaScript code can be built up and shared amongst users.

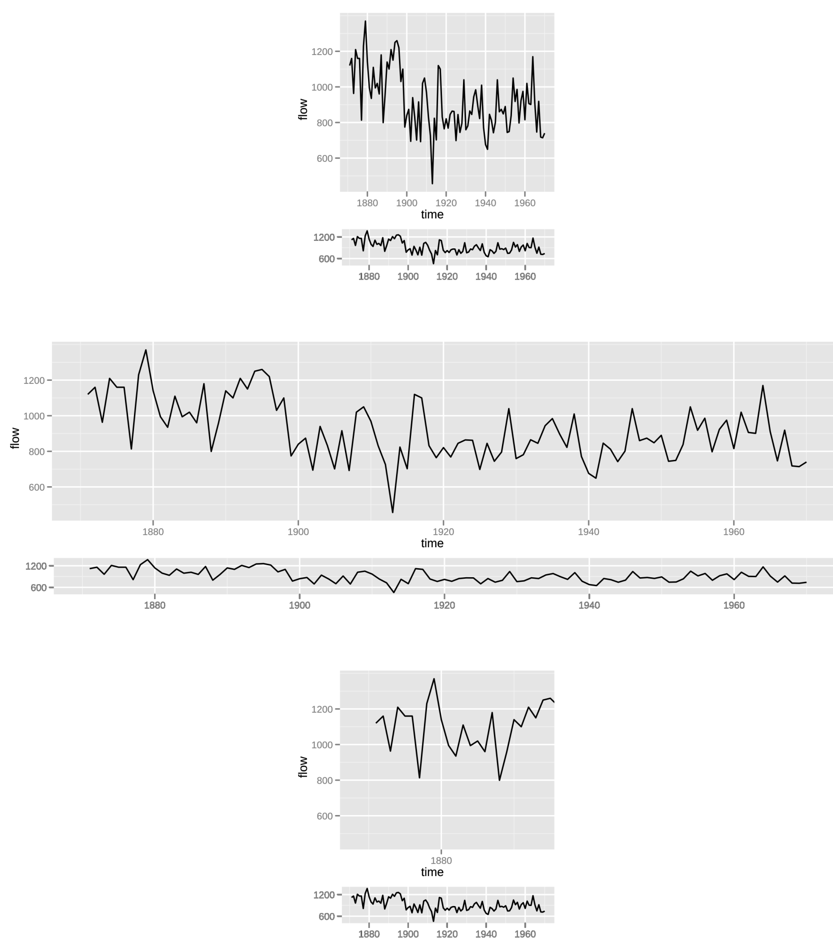

Figure 23: The top plot shows a **ggplot2** scene consisting of two plots, one above the other. The middle plot shows the same scene, just drawn much wider. The bottom plot shows a modification of the top scene, with the original plot region replaced with the plot region from the middle scene.

It should also be possible to build a layer of canned dynamic or interactive plot templates as higher-level R functions.

There are some features still missing from the package. A significant one is a lack of support for mathematical formulas (see `?plotmath`). It is hoped that it will be possible to add support based on the MathJax library (Cervone 2010).

It is also arguably a weakness that **gridSVG** only provides low-level infrastructure for creating animated and interactive plots; **gridSVG** does not contain any high-level functions that generate complete animated or interactive plots.

Finally, because fonts are not embedded in the SVG document, the positioning of text in a scene may not be exact.

6. Summary

The **gridSVG** package converts any **grid** scene to an SVG document. The `grid.hyperlink()` function allows a hyperlink to be associated with any component of the scene, the `grid.animate()` function can be used to animate any component of a scene, and the `grid.garnish()` function can be used to add SVG attributes to the components of a scene. By setting event handler attributes on a component, plus possibly using the `grid.script()` function to add JavaScript to the scene, it is possible to make the component respond to user input such as mouse clicks.

References

Bostock M, Heer J (2009). "Protovis: A Graphical Toolkit for Visualization." *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*. URL `http://vis.stanford.edu/papers/protovis`.

Cervone DP (2010). "MathJax: a JavaScript-based engine for including TEX and MathML in HTML." 2010 Joint Mathematics Meetings, URL `http://www.mathjax.org/`.

Chatzimichali E, Bessant C (2011). ***iWebPlots**: Interactive Web-Based Plots.* R package version 1.0-1, URL `http://CRAN.R-project.org/package=iWebPlots`.

Conway S (2010). ***webvis**: Create Graphics for the Web from R.* R package version 0.0.1, URL `http://CRAN.R-project.org/package=webvis`.

Eisenberg J (2002). *SVG Essentials*. O'Reilly Media, Inc, Sebastopol, CA.

Flanagan D (2006). *JavaScript: The Definitive Guide*. O'Reilly Media, Inc, Sebastopol, CA.

Gesmann M, de Castillo D (2011). ***googleVis**: Interface between R and the Google Visualisation API.* R package version 0.2.10, URL `http://CRAN.R-project.org/package=googleVis`.

Harold ER, Means WS (2004). *XML in a Nutshell*. O'Reilly Media, Inc, Sebastopol, CA.

Hothorn T, Zeileis A (2011). ***partykit**: A Toolkit for Recursive Partytioning.* R package version 0.1-1, URL `http://CRAN.R-project.org/package=partykit`.

Lawrence M, Temple Lang D (2010). ***RGtk2**: R Bindings for Gtk 2.8.0 and Above.* R package version 2.12.18, URL http://CRAN.R-project.org/package=RGtk2.

Meyer D, Zeileis A, Hornik K (2010). ***vcd**: Visualizing Categorical Data.* R package version 1.2-9, URL http://CRAN.R-project.org/package=vcd.

Murrell P (2011). *R Graphics.* 2 edition. CRC Press. ISBN 9781439831762.

Plate T (2011). ***RSVGTipsDevice**: An R SVG Graphics Device with Dynamic Tips and Hyperlinks.* R package version 1.0-2, URL http://CRAN.R-project.org/package=RSVGTipsDevice.

R Development Core Team (2011). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org/.

Rosling H (2008). "Gapminder: World." URL http://www.gapminder.org/world.

Rowlingson B (2008). ***imagemap**: Create HTML Imagemaps.* R package version 0.9-1/r6, URL http://R-Forge.R-project.org/projects/imagemap/.

Sarkar D (2008). ***Lattice**: Multivariate Data Visualization with R.* Springer-Verlag, New York. ISBN 978-0-387-75968-5, URL http://lmdvr.r-forge.r-project.org.

Simpson J (2002). *XPath and XPointer: Locating Content in XML Documents.* O'Reilly Media, Inc, Sebastopol, CA.

Swayne D, Cook D, Temple Lang D, Buja A (2010). "GGobi Software, Version 2.1." URL http://www.ggobi.org.

Temple Lang D (2010). ***XML**: Tools for Parsing and Generating XML within R and S-PLUS.* R package version 3.2-0, URL http://CRAN.R-project.org/package=XML.

Temple Lang D (2011). ***SVGAnnotation**: Tools for Post-Processing SVG Plots Created in R.* R package version 0.9-0.

Urbanek S, Horner J (2011). ***Cairo**: R Graphics Device Using Cairo Graphics Library for Creating High-Quality Bitmap, Vector, and Display Output.* R package version 1.4-9, URL http://CRAN.R-project.org/package=Cairo.

Urbanek S, Wichtrey T (2010). ***iplots**: iPlots - Interactive Graphics for R.* R package version 1.1-3, URL http://www.iPlots.org/.

Verzani J (2011). ***gWidgets**: gWidgets API for Building Toolkit-Independent, Interactive GUIs.* R package version 0.0-43, URL http://CRAN.R-project.org/package=gWidgets.

Viegas FB, Wattenberg M, van Ham F, Kriss J, McKeon M (2007). "ManyEyes: a Site for Visualization at Internet Scale." *IEEE Transactions on Visualization and Computer Graphics*, **13**, 1121–1128. ISSN 1077-2626.

Wickham H (2009). ***ggplot2**: Elegant Graphics for Data Analysis.* Springer-Verlag, New York. ISBN 978-0-387-98140-6. URL http://had.co.nz/ggplot2/book.

Wild CJ, Pfannkuch M, Regan M, Horton NJ (2011). "Towards More Accessible Conceptions of Statistical Inference." *Journal of the Royal Statistical Society: Series A (Statistics in Society)*, **174**(2), 247–295. ISSN 1467-985X.

Xie Y (2011). ***animation**: A Gallery of Animations in Statistics and Utilities to Create Animations*. R package version 2.0-3, URL http://CRAN.R-project.org/package=animation.

Affiliation:

Paul Murrell
Department of Statistics
The University of Auckland
38 Princes Street, Auckland
New Zealand
E-mail: paul@stat.auckland.ac.nz
URL: http://www.stat.auckland.ac.nz/~paul/

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/

Volume VV, Issue II *Submitted:* yyyy-mm-dd
MMMMMM YYYY *Accepted:* yyyy-mm-dd